

**TRANSPORTATION ANALYSIS SIMULATION SYSTEM  
(TRANSIMS)**

**VERSION 1.0**

**Network Subsystem for IOC-1**

**B. W. Bush**  
**Energy and Environment Analysis Group**  
**Los Alamos National Laboratory**

**K. P. Berkbigler**  
**Computer Research and Applications Group**  
**Los Alamos National Laboratory**

**March 1998**

**[Tab 5]**



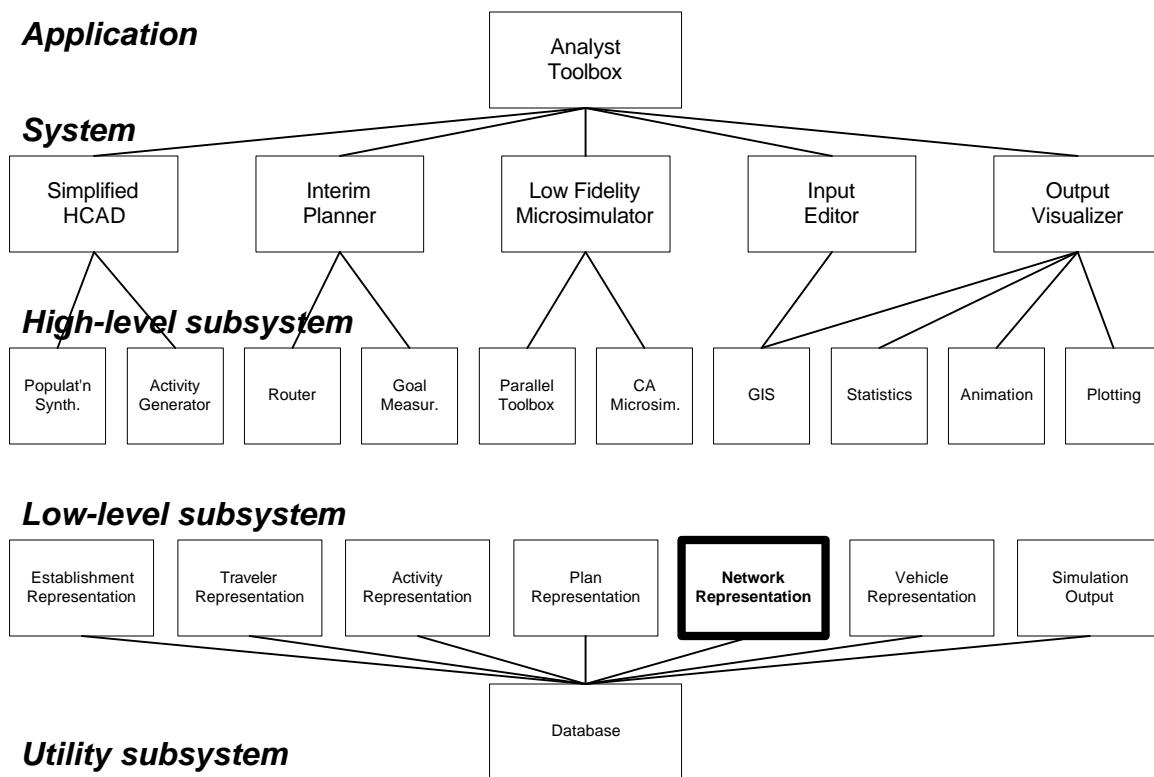
## Contents

<b>1. INTRODUCTION.....</b>	<b>5</b>
<b>2. DESIGN .....</b>	<b>6</b>
2.1 CONCEPTS.....	6
2.1.1 Node .....	6
2.1.2 Link .....	6
2.1.3 Lane.....	6
2.1.4 Pocket Lane.....	6
2.1.5 Parking .....	6
2.1.6 Traffic Control .....	7
2.1.7 Lane Connectivity .....	7
2.1.8 Unsignalized Node .....	7
2.1.9 Signalized Node .....	7
2.1.10 Phasing Plan .....	7
2.1.11 Timing Plan .....	7
2.1.12 Study Areas and Buffer Areas.....	7
2.2 CLASSES .....	8
2.2.1 TNetFactory .....	10
2.2.2 TNetNetwork.....	11
2.2.3 TNetSubnetwork .....	11
2.2.4 TNetReader.....	12
2.2.5 TNetNode.....	12
2.2.6 TNetNodeReader.....	13
2.2.7 TNetLink .....	14
2.2.8 TNetLinkReader.....	16
2.2.9 TNetLocation .....	17
2.2.10 TNetLane .....	18
2.2.11 TNetLaneLocation.....	19
2.2.12 TNetLaneConnectivityReader.....	19
2.2.13 TNetAccessory .....	20
2.2.14 TNetAccessoryReader.....	20
2.2.15 TNetPocket.....	21
2.2.16 TNetPocketReader.....	21
2.2.17 TNetParking.....	22
2.2.18 TNetParkingReader.....	22
2.2.19 TNetTrafficControl.....	23
2.2.20 TNetNullControl .....	24
2.2.21 TNetIsolatedControl .....	25
2.2.22 TNetUnsignalizedControl .....	25
2.2.23 TNetUnsignalizedControlReader .....	26
2.2.24 TNetSignalizedControl.....	27
2.2.25 TNetSignalizedControlReader .....	28
2.2.26 TNetTimedControl .....	29
2.2.27 TNetPhase .....	30
2.2.28 TNetPhaseDescription .....	30
2.2.29 TNetPhasingPlan.....	32
2.2.30 TNetPhasingPlanReader .....	32
2.2.31 TNetTimingPlanReader.....	33
2.2.32 TNetSignalCoordinator .....	34

2.2.33 TNetSimulationArea .....	35
2.2.34 TNetSimulationAreaReader.....	36
2.2.35 TNetSimulationAreaLinkReader .....	36
2.2.36 TGeoPoint .....	36
2.2.37 TGeoRectangle .....	37
2.2.38 TGeoFilterFunction .....	37
2.2.39 TGeoFilterNone.....	38
2.2.40 TGeoFilterRectangle.....	38
2.2.41 TNetException .....	38
<b>3. IMPLEMENTATION.....</b>	<b>40</b>
3.1 C++ LIBRARIES .....	40
3.2 SOURCES FOR TRAFFIC ENGINEERING INFORMATION.....	40
<b>4. USAGE.....</b>	<b>41</b>
4.1 ACCESSING NETWORK DATA VIA C++.....	41
4.2 NETWORK DATA TABLES .....	42
4.2.1 File Formats .....	42
4.2.2 Example .....	47
<b>5. FUTURE WORK.....</b>	<b>56</b>
<b>6. REFERENCES .....</b>	<b>57</b>

# 1. INTRODUCTION

The TRANSIMS network representation provides access to detailed information about streets, intersections, and signals in a road network. It forms a layer separating the other subsystems from the actual network data tables so that the other subsystems do not need to access the data tables directly or deal with the format and organization of the tables. **Figure 1** shows the position of the network subsystem within the TRANSIMS software architecture. This subsystem depends only upon the database subsystem, and only upon that during network object construction from data tables.



**Figure 1: Location of the Network Subsystem in the TRANSIMS Software Architecture**

This subsystem allows the user to construct multiple subnetworks from the network database tables. It includes road network objects such as nodes (intersections), links (road/street segments), lanes, and traffic controls (signs and signals). Link attributes for the road network include such characteristics as link type, length, directionality, speed limit, number of lanes, grade, and intersection setbacks. Traffic controls may be either signs (stop and yield) or pre-timed signals.

The body of this document outlines the design, implementation, and usage of the subsystem.

## **2. DESIGN**

### **2.1 Concepts**

#### **2.1.1 Node**

A node is the part of the network corresponding to a *vertex* in graph theory. Nodes typically occur at intersections in the road network. A node must be present where the network branches and where the permanent number of lanes changes. A lane is considered *permanent* if it is not a temporary pocket lane (see the definition of pocket lane below). A node may be present where neither of the aforementioned occurs, however. Nodes are not required where turn pockets start, as these are not considered permanent lanes. Each node has a traffic control associated with it (null, unsignalized, pre-timed, actuated, coordinated, etc.).

#### **2.1.2 Link**

A link is the part of the network corresponding to an *edge* in graph theory. Links represent street and road segments. Each link has a constant number of permanent lanes, but may have a variable number pocket lanes. A link may have lanes in both directions, or the lanes in opposite directions may be on separate links (in which case no passing into oncoming lanes will be possible).

#### **2.1.3 Lane**

A lane is where traffic flows on a link. The lanes on each side/direction of the link are numbered separately, starting with lane number one as the leftmost lane (relative to the direction of travel). Each successive lane to the right of it is numbered one greater than its predecessor. Pocket lanes (i.e., turn pockets, merges, and pull-outs) are numbered in sequence, even if they do not exist for the full length of the link. A two-way left-turn lane, if present, is considered to be lane number zero.

#### **2.1.4 Pocket Lane**

A pocket lane is either (a) a right or left-turn pocket (a lane that starts after the *from* intersection and ends at the *to* intersection), (b) a right or left pull-out (a lane that starts after the *from* intersection and ends before the *to* intersection), or (c) a right or left merge pocket (a lane that starts at the *from* intersection and ends before the *to* intersection). If a lane starts at the *from* intersection and ends at the *to* intersection, it is considered a permanent lane, not a pocket lane.

#### **2.1.5 Parking**

Parking areas are located along links and are used as origins and destinations for vehicle trips. Parking may be placed where it actual is physically located in the network, or it may be placed in aggregate *generic* parking areas representing several of the driveways, lots, parking places, etc., on a link. Places where vehicles leave the network are called *boundary* parking areas.

## **2.1.6 Traffic Control**

Each node has a traffic control associated with it. The traffic control specifies how lanes are connected across the node and the type of sign or signalized control that determines who has the right-of-way.

## **2.1.7 Lane Connectivity**

Lane connectivity specifies how lanes are connected across a node. Lanes are numbered from the median and include turn pockets. Incoming and outgoing links and lanes are defined relative to the node. For each incoming lane on an incoming link, at least one outgoing lane must be specified for each outgoing link that a vehicle on the incoming link can transition to. Multiple outgoing lanes may be defined for an outgoing link, if desired.

## **2.1.8 Unsignalized Node**

An unsignalized node represents the type of sign control, if any, that is present at an unsignalized node. Examples are stop and yield signs. Nodes where only the number of permanent lanes is changing are generally considered unsignalized.

## **2.1.9 Signalized Node**

A signalized node represents a traffic light. Each signal has a timing plan and a phasing plan.

### **2.1.10 Phasing Plan**

A phasing plan specifies the turn protection in effect for transitioning from an incoming link to an outgoing link during a particular phase of a specific timing plan.

### **2.1.11 Timing Plan**

A timing plan specifies the lengths of the intervals during the specific phases for a traffic light. Many nodes may have the same timing plan. It is possible for each phase to transition to more than one phase if required.

### **2.1.12 Study Areas and Buffer Areas**

The microsimulation distinguishes two types of links in its calculations: Study area links are the links of interest for the traffic analyst. The output subsystem, for instance, records events such as when a vehicle leaves or enters the study area. The nature of the microsimulation makes it necessary also to simulate traffic on additional buffer area links. Typically, these links form a *fringe* about two links thick around the study area. A simulation includes buffer links in order to avoid edge effects such as when vehicles enter the study area on its boundary; the *buffer* gives these vehicles time to interact with other traffic and achieve realistic behavior before entering the study area.

## 2.2 Classes

The network subsystem has classes for constructing a road network from database tables and for supplying information about the road network to clients. Figure 2 through Figure 6 show the relationships between the classes.

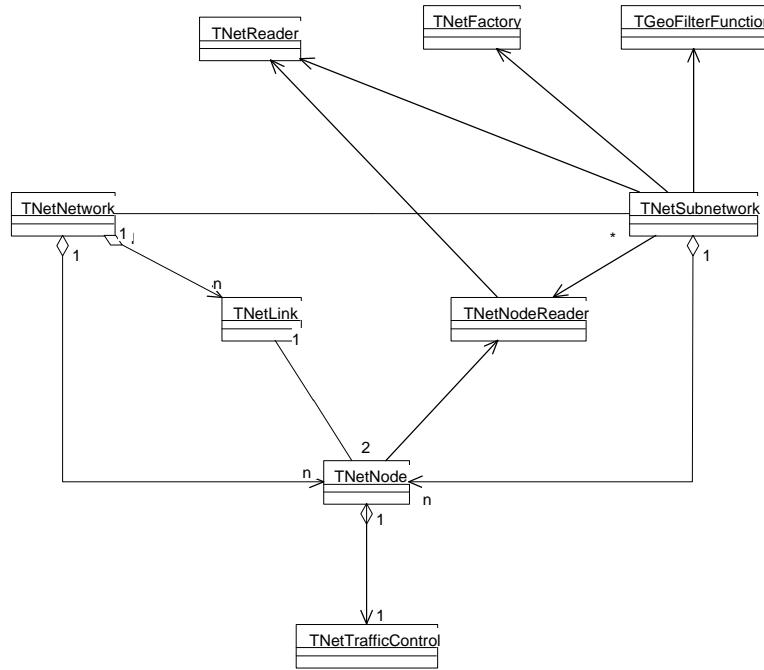


Figure 2: Class Diagram for Node-related Classes (unified notation)

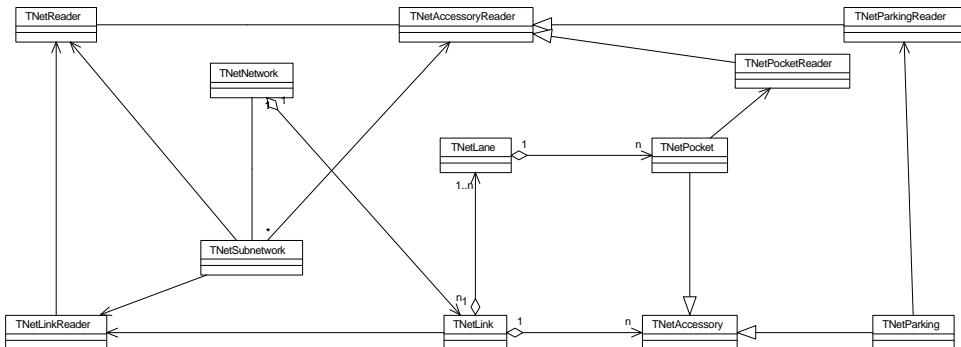
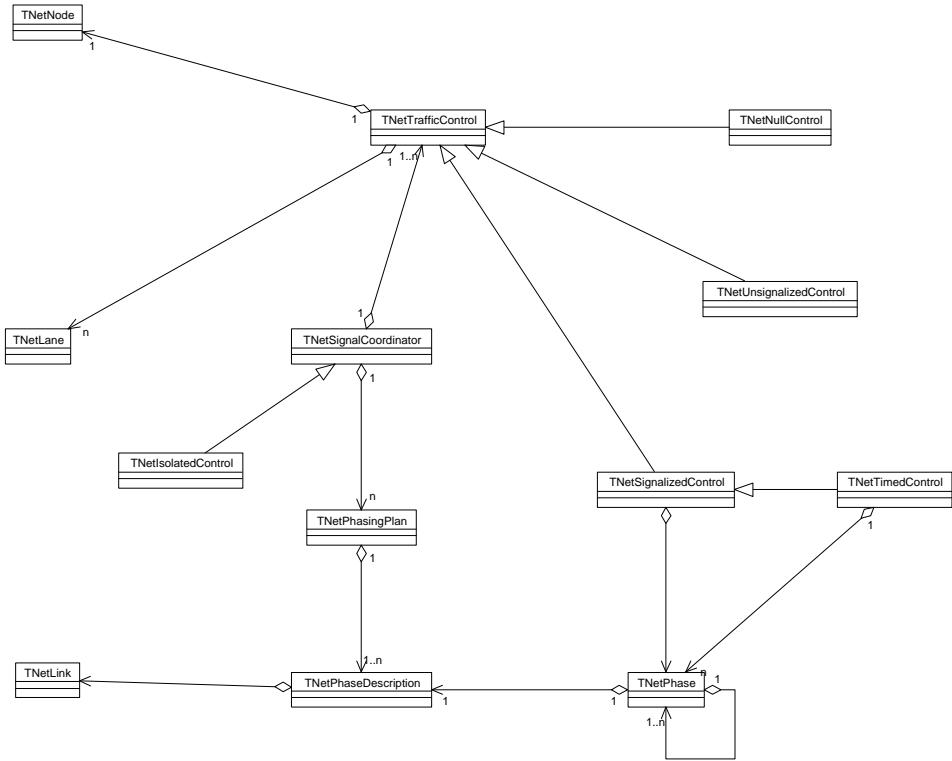
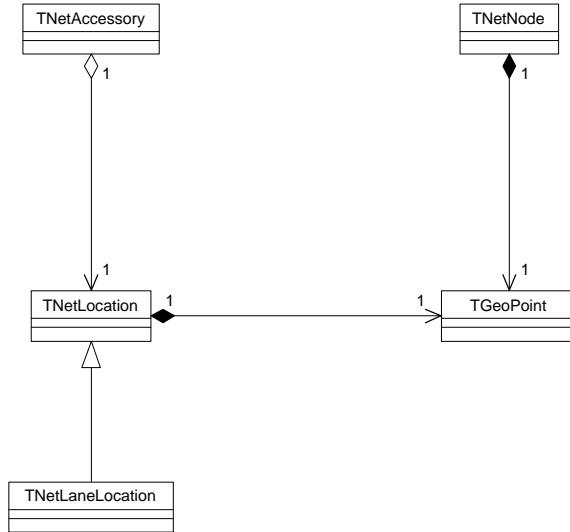


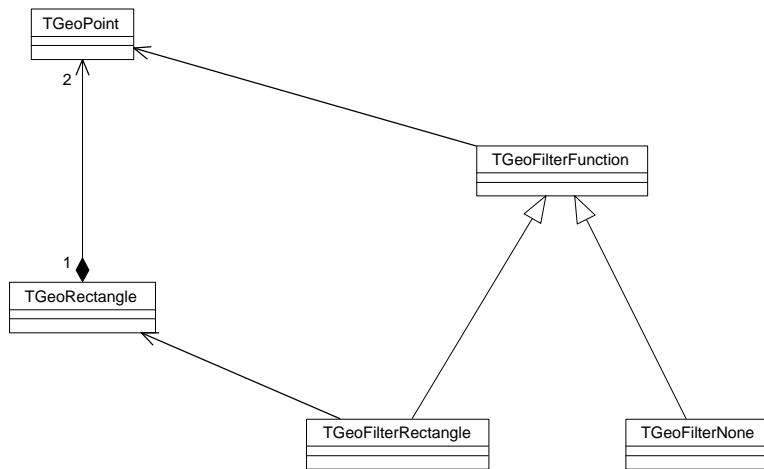
Figure 3: Class Diagram for Link-related Classes (unified notation)



**Figure 4: Class Diagram for Control-related Classes (unified notation)**



**Figure 5: Class Diagram for Location-related Classes (unified notation)**



**Figure 6: Class Diagram for Geography-related Classes (unified notation)**

### 2.2.1 TNetFactory

A network factory allocates and constructs new network objects.

```

virtual TNetNode* NewNode(TNetNodeReader& reader)
    Return a new node from the specified reader.

virtual TNetLink* NewLink(TNetLinkReader& reader)
    Return a new link from the specified reader.

virtual TNetPocket* NewPocket(TNetPocketReader& reader)
    Return a new pocket from the specified reader.

virtual TNetParking* NewParking(TNetParkingReader& reader)
    Return a new parking place from the specified reader.

virtual TNetUnsignalizedControl*
    NewUnsignalizedControl(TNetUnsignalizedControlReader&
                           reader, TNetNetwork& network)
    Return a new unsignalized control from the specified reader and for the specified network.

virtual TNetTimedControl*
    NewTimedControl(TNetSignalizedControlReader& reader,
                   TNetNetwork& network)
    Return a new timed control from the specified reader and for the specified network.

virtual TNetIsolatedControl* NewIsolatedControl(NetNodeId id)
    Return a new isolated control for the specified node.

virtual TNetNullControl* NewNullControl(TNetNode& node)
    Return a new null control for the specified node.
  
```

## 2.2.2 TNetNetwork

A network represents all of the network database that has been instantiated. Each network manages/owns the nodes, links, and signal coordinators it contains and has a map of parking places.

```
TNetNetwork()
Construct a network.
```

```
NodeMap& GetNodes()
const NodeMap& GetNodes() const
Return the set of nodes in the network.
```

```
LinkMap& GetLinks()
const LinkMap& GetLinks() const
Return the set of links in the network.
```

```
ParkingMap& GetParkings()
const ParkingMap& GetParkings() const
Return the set of parking places in the network.
```

```
SignalCoordinatorMap& GetSignalCoordinators()
const SignalCoordinatorMap& GetSignalCoordinators() const
Return the set of signal coordinators in the network.
```

## 2.2.3 TNetSubnetwork

A subnetwork represents a subset of the instantiated network. Each subnetwork is part of a network and has references to the nodes and links it contains.

```
TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
               TGeoFilterFunction& filter = TGeoFilterNone(),
               TNetFactory& factory = TNetFactory())
Construct a subnetwork with geographic filtering using the reader.
```

```
TNetSubnetwork(TNetReader& reader, TNetNetwork& network,
               NodeIdSet& nodesRequested, LinkIdSet& linksRequested,
               NodeSet& nodesProvided, LinkSet& linksProvided,
               TNetFactory& factory = TNetFactory())
Construct a subnetwork of specified nodes and links using the reader.
```

```
NodeSet& GetNodes()
const NodeSet& GetNodes() const
Return the set of nodes in the subnetwork.
```

```
LinkSet& GetLinks()
const LinkSet& GetLinks() const
Return the set of links in the subnetwork.
```

```
TNetNetwork& GetNetwork()
const TNetNetwork& GetNetwork() const
    Return the network.
```

## 2.2.4 TNetReader

A network reader reads a network from the database. Each reader has a node, link, pocket, parking, lane connectivity, unsignalized control, signalized control, phasing plan, and timing plan table.

```
TNetReader(TDbTable nodeTable, TDbTable linkTable, TDbTable
           pocketTable, TDbTable parkingTable, TDbTable
           connTable, TDbTable unsigTable, TDbTable sigTable,
           TDbTable phaseTable, TDbTable timeTable)
Construct a reader for the specified tables.
```

```
TDbTable& GetNodeTable()
    Return the node table.
```

```
TDbTable& GetLinkTable()
    Return the link table.
```

```
TDbTable& GetPocketTable()
    Return the pocket table.
```

```
TDbTable& GetParkingTable()
    Return the parking table.
```

```
TDbTable& GetLaneConnectivityTable()
    Return the lane connectivity table.
```

```
TDbTable& GetUnsignalizedControlTable()
    Return the unsignalized control table.
```

```
TDbTable& GetSignalizedControlTable()
    Return the signalized control table.
```

```
TDbTable& GetPhasingPlanTable()
    Return the phasing plan table.
```

```
TDbTable& GetTimingPlanTable()
    Return the timing plan table.
```

## 2.2.5 TNetNode

A node is the part of the network corresponding to a *vertex* in graph theory. A node must be present where the network branches and where the permanent number of lanes changes. (A node may be present where neither of the aforementioned occurs, however.) Each node has a unique id, a geographic position, and an associated traffic control, and it is connected to links.

```

TNetNode( TNetNodeReader& reader )
    Construct a node using the reader.

TNetNode( NetNodeId id )
    Construct a dummy node with the specified id.

NetNodeId GetId() const
    Return the id of the node.

TGeoPoint& GetGeographicPosition()
const TGeoPoint& GetGeographicPosition() const
    Return the geographic position of the node.

LinkRing& GetLinks()
const LinkRing& GetLinks() const
    Return the ring of links in order.

void AddLink(TNetLink* link)
    Add the specified link to the ring of links.

TNetTrafficControl& GetTrafficControl()
const TNetTrafficControl& GetTrafficControl() const
    Return the traffic control for the node.

void SetTrafficControl (TNetTrafficControl*)
    Define the traffic control for the node.

bool operator==(const TNetNode& node) const
    Return whether the node has the same id as the given node.

bool operator!=(const TNetNode& node) const
    Return whether the node has a different id from the given node.

```

## 2.2.6 TNetNodeReader

A node reader reads node values from the database. Each node reader has a database table accessor and database fields.

```

TNetNodeReader( TNetReader& reader )
    Construct a node reader for a given network.

TNetNodeReader( const TNetNodeReader& reader )
    Construct a copy of the given node reader.

TNetNodeReader& operator=( const TNetNodeReader& reader )
    Make the reader a copy of the given node reader.

void Reset()
    Reset the iteration over the table.

```

```

void GetNextNode()
    Get the next node in the table.

bool MoreNodes() const
    Return whether there are any more nodes in the table.

NetNodeId GetId() const
    Return the id for the current node.

TGeoPoint GetGeographicPosition() const
    Return the geographic position for the current node.

```

### 2.2.7 TNetLink

A link is the part of the network corresponding to an *edge* in graph theory. Each link has a constant number of permanent lanes, but may have turn pocket lanes also. A link may have lanes in both directions, or the lanes in opposite directions may be on separate links (in which case no passing into oncoming lanes will be possible). Each link has a unique id, nodes at its two ends, zero or more accessories, zero or more lanes on each side, zero or more permanent lanes on each side, a length, setback distances from the intersection, through links at either end, speed limits in either direction, and an angle (in radians) from its endpoints.

```

enum EFunctionalClass {kOther}
    There are several functions for a link.

TNetLink(TNetLinkReader& reader)
    Construct a link using the reader.

TNetLink(NetLinkId id)
    Construct a dummy link with the specified id.

NetLinkId GetId() const
    Return the id of the link.

void GetNodes(NetNodeId& nodeA, NetNodeId& nodeB) const
void GetNodes(TNetNode*& nodeA, TNetNode*& nodeB) const
void GetNodes(const TNetNode*& nodeA, const TNetNode*& nodeB)
    const
    Return the nodes at the ends of the link.

void SetNodes(TNetNode* nodeA, TNetNode* nodeB)
    Set the nodes at the ends of the link.

AccessoryCollection& GetAccessories()
const AccessoryCollection& GetAccessories() const
    Return the accessories on the link.

```

```
LaneCollection& GetLanesFrom(const TNetNode& node, bool pockets  
    = FALSE)
```

```
const LaneCollection& GetLanesFrom(const TNetNode& node, bool  
    pockets = FALSE) const
```

Return the lanes going away from the specified node. The lanes are ordered (but not necessarily numbered) from the divider outward. Pockets are included optionally. A TNetNotFound exception is thrown if the node is not at one of the link's ends.

```
LaneCollection& GetLanesTowards(const TNetNode& node, bool  
    pockets = FALSE)
```

```
const LaneCollection& GetLanesTowards(const TNetNode& node, bool  
    pockets = FALSE) const
```

Return the lanes going toward the specified node. The lanes are ordered (but not necessarily numbered) from the divider outward. Pockets are included optionally. A TNetNotFound exception is thrown if the node is not at one of the link's ends.

```
REAL GetLength(bool setback = FALSE) const
```

Return the length of the link (using the default units). The length is measured from one node to the other, unless setback distance subtraction is requested.

```
REAL GetSetback(const TNetNode& node) const
```

Return the setback distance (using the default units) at the specified node. A TNetNotFound exception is thrown if the node is not at one of the link's ends.

```
NetLinkId GetThroughLink(const TNetNode& node) const
```

Return the through link at the given node on the current link. A TNetNotFound exception is thrown if the node is not at one of the link's ends.

```
REAL GetSpeedLimitTowards(const TNetNode& node) const
```

Return the speed limit of the lanes heading toward the given node on the current link. A TNetNotFound exception is thrown if the node is not at one of the link's ends.

```
REAL GetAngle(const TNetNode& node) const
```

Return the angle (in radians) of the link from the specified node. A TNetNotFound exception is thrown if the node is not at one of the link's ends.

```
TNetNode& GetNodeBetween(const TNetLink& link) const
```

Return the node between the current link and the given link. A TNetNotFound exception is thrown if the links are not adjacent.

```
bool operator==(const TNetLink& link) const
```

Return whether the link has the same id as the given link.

```
bool operator!=(const TNetLink& link) const
```

Return whether the link has a different id from the given link.

## 2.2.8 TNetLinkReader

A link reader reads link values from the database. Each link reader has a database table accessor and database fields.

```
TNetLinkReader(TNetReader& reader)
    Construct a link reader for a given network.

TNetLinkReader(const TNetLinkReader& reader)
    Construct a copy of the given link reader.

TNetLinkReader& operator=(const TNetLinkReader& reader)
    Make the reader a copy of the given link reader.

void Reset()
    Reset the iteration over the table.

void GetNextLink()
    Get the next link in the table.

bool MoreLinks() const
    Return whether there are any more links in the table.

NetLinkId GetId() const
    Return the id for the current link.

void GetNodeIds(NetNodeId& nodeA, NetNodeId& nodeB) const
    Return the node ids at the ends of the current link.

BYTE GetPermanentLaneCountTowards(NetNodeId id) const
    Return the number of permanent lanes heading toward the given node on the current link.
    A TNetNotFound exception is thrown if the node is not at one of the link's ends.

BYTE GetLeftPocketLaneCountTowards(NetNodeId id) const
    Return the number of left pocket lanes heading toward the given node on the current link.
    A TNetNotFound exception is thrown if the node is not at one of the link's ends.

BYTE GetRightPocketLaneCountTowards(NetNodeId id) const
    Return the number of right pocket lanes heading toward the given node on the current link.
    A TNetNotFound exception is thrown if the node is not at one of the link's ends.

bool HasTwoWayLeftTurnLane() const
    Return whether there is a two-way left-turn lane on the current link.

REAL GetLength() const
    Return the length of the current link.
```

```

REAL GetGradeTowards(NetNodeId id) const
    Return the percent grade of the lanes heading toward the given node on the current link. A
    TNetNotFound exception is thrown if the node is not at one of the link's ends.

REAL GetSetbackDistance(NetNodeId id) const
    Return the setback distances at the given node on the current link. A TNetNotFound
    exception is thrown if the node is not at one of the link's ends.

NetLinkId GetThroughLink(NetNodeId id) const
    Return the through link at the given node on the current link. A TNetNotFound
    exception is thrown if the node is not at one of the link's ends.

REAL GetCapacityTowards(NetNodeId id) const
    Return the capacity in vehicles-per-hour of the lanes heading toward the given node on the
    current link. A TNetNotFound exception is thrown if the node is not at one of the
    link's ends.

REAL GetSpeedLimitTowards(NetNodeId id) const
    Return the speed limit of the lanes heading toward the given node on the current link. A
    TNetNotFound exception is thrown if the node is not at one of the link's ends.

REAL GetFreeFlowSpeedTowards(NetNodeId id) const
    Return the free-flow speed of the lanes heading toward the given node on the current link.
    A TNetNotFound exception is thrown if the node is not at one of the link's ends.

REAL GetCrawlSpeedTowards(NetNodeId id) const
    Return the crawl speed of the lanes heading toward the given node on the current link. A
    TNetNotFound exception is thrown if the node is not at one of the link's ends.

TNetLink::EFunctionalClass GetFunctionalClass() const
    Return the functional class for the link.

UINT GetCostTowards(NetNodeId id) const
    Return the cost (in arbitrary units) for traveling on the link toward the given node. A
    TNetNotFound exception is thrown if the node is not at one of the link's ends.

```

## 2.2.9 TNetLocation

A location is a point along a link. Each location is on a link, is measured from a node, and is specified as an offset relative to the start of its link.

```

TNetLocation(TNetLink& link, TNetNode& node, REAL offset)
    Construct a location along the given node and link with the given offset from the node.
    The exception TNetNotFound is thrown if the node is not at one of the link's ends.

TNetLocation(const TNetLocation& location)
    Construct a copy of the given location.

```

```

TNetLocation& operator=(const TNetLocation& location)
    Make the location a copy of the given location.

TNetLink& GetLink()
const TNetLink& GetLink() const
    Return the link on which the location lies.

REAL GetOffsetFrom(const TNetNode& node) const
    Return the distance from the given endpoint of the link. The exception TNetNotFound
    is thrown if the node is not at one of the link's ends.

TGeoPoint GetGeographicPosition() const
    Return the geographic position of the location.

virtual bool IsOnSpecificLane() const
    Return whether the location is lane-specific.

```

## 2.2.10 TNetLane

A lane is where traffic flows on a link. Each lane belongs on a link and has a starting node, a number, and zero or more pockets.

```

TNetLane(TNetLink& link, NetNodeId node, NetLaneNumber number)
    Construct a lane on the specified link, starting at the specified node, and identified by the
    specified number.

TNetLink& GetLink()
const TNetLink& GetLink() const
    Return the link on which the lane lies.

TNetNode& GetStartNode()
const TNetNode& GetStartNode() const
    Return the starting node for the lane.

TNetNode& GetEndNode()
const TNetNode& GetEndNode() const
    Return the end node for the lane.

PocketCollection& GetPockets()
const PocketCollection& GetPockets() const
    Return the pockets on the lane.

const TNetLane* GetLeftAdjacentLane() const
    Return the lane to the left, if any.

const TNetLane* GetRightAdjacentLane() const
    Return the lane to the right, if any.

NetLaneNumber GetNumber() const
    Return the number of the lane.

```

## 2.2.11 TNetLaneLocation

A location is a lane-specific point along a link. Each lane location is on a lane.

```
TNetLaneLocation(TNetLane& lane, REAL offset)
    Construct a location along the lane with the given offset from its start.

TNetLaneLocation(const TNetLaneLocation& location)
    Construct a copy of the given location.

TNetLaneLocation& operator=(const TNetLaneLocation& location)
    Make the location a copy of the given location.

TNetLane& GetLane()
const TNetLane& GetLane() const
    Return the lane for the location.

bool IsOnSpecificLane() const
    Return whether the location is lane-specific.
```

## 2.2.12 TNetLaneConnectivityReader

This reader reads lane connectivity values from the database. Each lane connectivity has a database table accessor and database fields.

```
TNetLaneConnectivityReader(TNetReader& reader)
    Construct a lane connectivity reader for a given network.

TNetLaneConnectivityReader(const TNetLaneConnectivityReader&
                           reader)
    Construct a copy of the given lane connectivity reader.

TNetLaneConnectivityReader& operator=(const
                                         TNetLaneConnectivityReader& reader)
    Make the reader a copy of the given lane connectivity reader.

void Reset()
    Reset the iteration over the table.

void GetNextNode()
    Get the next node in the table.

bool MoreNodes()
    Return whether there are any more nodes in the table.

NetNodeId GetNode() const
    Return the id for the current node.

NetLinkId GetInlink() const
    Return the id for incoming link.
```

```
NetLaneNumber GetInlane() const  
    Return the id for incoming lane.
```

```
NetLinkId GetOutlink() const  
    Return the id for outgoing link.
```

```
NetLaneNumber GetOutlane() const  
    Return the id for outgoing lane.
```

### 2.2.13 TNetAccessory

An accessory is where something happens along a link. Each accessory has a unique id, a type, and a location.

```
enum EType {kPocket, kParking}  
    There are several types of accessories.
```

```
TNetAccessory(NetAccessoryId id, EType type)  
    Construct an accessory with the specified id and type.
```

```
NetAccessoryId GetId() const  
    Return the id of the accessory.
```

```
EType GetType() const  
    Return the type of the accessory.
```

```
TNetLocation& GetLocation()  
const TNetLocation& GetLocation() const  
    Return the location of the accessory.
```

```
void SetLocation(const TNetLocation& location)  
    Set the location of the accessory.
```

### 2.2.14 TNetAccessoryReader

An accessory reader reads accessory values from the database. Each accessory has a database table accessor and database fields.

```
TNetAccessoryReader(TDbTable& table)  
    Construct an accessory reader for a given network.
```

```
TNetAccessoryReader(const TNetParkingReader& reader)  
    Construct a copy of the given accessory reader.
```

```
TNetAccessoryReader& operator=(const TNetParkingReader& reader)  
    Make the accessory reader a copy of the given accessory reader.
```

```
void Reset()  
    Reset the iteration over the table.
```

```

void GetNextAccessory()
    Get the next accessory in the table.

bool MoreAccessories() const
    Return whether there are any more accessories in the table.

NetAccessoryId GetId() const
    Return the id for the current accessory.

void GetLocation(NetLinkId& link, NetNodeId& node, REAL& offset)
    const
    Return the location of the current accessory.

```

## 2.2.15 TNetPocket

A pocket is a length of lane intended for special uses such as buses and pulling out, vehicles waiting for turns, vehicles accelerating in order to merge, etc. Each pocket has a style and a length.

```

enum EStyle {kTurn, kPullout, kMerge}
    There are several styles of pockets.

TNetPocket(TNetPocketReader& reader)
    Construct the pocket accessory from the specified reader.

EStyle GetStyle() const
    Return the style of the pocket.

REAL GetLength() const
    Return the length of the pocket.

TNetLane& GetLane()
const TNetLane& GetLane() const
    Return the lane for the pocket.

```

## 2.2.16 TNetPocketReader

A pocket reader reads pocket values from the database. Each pocket has database table accessor and database fields.

```

TNetPocketReader(TNetReader& reader)
    Construct a pocket reader for a given network.

TNetPocketReader(const TNetPocketReader& reader)
    Construct a copy of the given pocket reader.

TNetPocketReader& operator=(const TNetPocketReader& reader)
    Make the pocket reader a copy of the given pocket reader.

```

```
NetLaneNumber GetLaneNumber() const  
    Return the lane number of the current pocket.
```

```
TNetPocket::EStyle GetStyle() const  
    Return the style of the current pocket.
```

```
REAL GetLength() const  
    Return the length of the current pocket.
```

## 2.2.17 TNetParking

A parking place is a source or sink of vehicles along a link. A parking place has a style and a capacity and may be generic.

```
enum EStyle {kParallelOnStreet, kHeadInOnStreet, kDriveway,  
            kLot, kBoundary}  
There are several styles of parking.
```

```
TNetParking(TNetParkingReader& reader)  
Construct the parking from the specified reader.
```

```
TNetParking (NetAccessoryId, TNetLink&, TNetNode&, REAL offset,  
            EStyle style, UINT capacity, bool generic)  
Construct a parking place with specified values.
```

```
EStyle GetStyle() const  
    Return the style of the parking.
```

```
UINT GetCapacity() const  
    Return the capacity of the parking.
```

```
bool IsGeneric() const  
    Return whether the parking is generic.
```

```
static NetAccessoryId GenerateId(const TNetNetwork&, NetLinkId)  
Generate a unique parking place id.
```

## 2.2.18 TNetParkingReader

A parking reader reads parking values from the database. Each parking reader has a database table accessor and database fields.

```
TNetParkingReader(TNetReader& reader)  
Construct a parking reader for a given network.
```

```
TNetParkingReader(const TNetParkingReader& reader)  
Construct a copy of the given parking reader.
```

```
TNetParkingReader& operator=(const TNetParkingReader& reader)  
Make the parking reader a copy of the given parking reader.
```

```

TNetParking::EStyle GetStyle() const
    Return the style of the current parking.

UINT GetCapacity() const
    Return the capacity of the current parking.

bool IsGeneric() const
    Return whether the current parking is generic.

```

## 2.2.19 TNetTrafficControl

A traffic control is associated with each node. The traffic control specifies how lanes are connected across the node and the type of sign or signalized control that determines who has the right-of-way. A traffic control knows the node with which it is associated, and it contains a table that describes how lanes are connected across the node.

```

enum ETrafficControl {kNone, kStop, kYield, kWait, kCaution,
                      kPermitted, kProtected}
    Vehicle signs and signals indicate right-of-way for movements.

TNetTrafficControl()
TNetTrafficControl(TNetNode& node)
TNetTrafficControl(const TNetTrafficControl& control)
    Construct a traffic control.

virtual void AllowedMovements (LaneCollection& lanes, const
                                TNetLink& fromlink, const TNetLane& fromlane, const
                                TNetLink& tolink)
    Return the lanes on next link to which transition from specified lane on this link can be
    made. Return an empty collection if transition is not possible.

virtual void AllowedMovements (LaneCollection& lanes, const
                                TNetLink& fromlink, const TNetLink& tolink, const
                                TNetLane& tolane)
    Return the lanes on this link from which transition to specified lane on next link can be
    made. Return an empty collection if transition is not possible.

virtual void AllowedMovements (LaneCollection& lanes, const
                                TNetLink& fromlink, const TNetLink& tolink, bool
                                phase = FALSE)
    Return the lanes ordered from median that may be used to transition from current link to
    next link.

virtual void InterferingLanes (LaneCollection& lanes, const
                               TNetLane& fromlane, const TNetLane& tolane, bool
                               phase = FALSE)
    Return the lanes that must be examined for interference when transitioning from current
    lane to next lane.

```

```

virtual ETrafficControl GetVehicleControl(const TNetLane& lane)
    const
Return the vehicle control signal for the specified lane.

TNetNode& GetNode()
const TNetNode& GetNode() const
Return the associated node.

void SetConnectivity(TNetLane& inlane, TNetLane& outlane)
Define the lane connectivity for the specified lane.

void SetConnectivity (TNetLaneConnectivityReader& reader,
                     TNetNetwork& network)
Define the lane connectivity using the reader.

TNetTrafficControl::ConnectedCollection& GetConnectivity(const
                                                          TNetLane& lane)
const TNetTrafficControl::ConnectedCollection&
      GetConnectivity(const TNetLane& lane) const
Return the lane connectivity for the specified lane.

TNetTrafficControl::ConnectivityMap& GetConnectivity()
const TNetTrafficControl::ConnectivityMap& GetConnectivity()
const
Return the lane connectivity map.

```

## 2.2.20 TNetNullControl

Null controls are used by nodes just outside the network boundary.

```

TNetNullControl()
TNetNullControl(TNetNode& node)
TNetNullControl(const TNetNullControl& control)
Construct a null traffic control.

TNetNullControl& operator=(const TNetNullControl& control)
Assign a null traffic control.

bool operator==(const TNetNullControl& control) const
bool operator!=(const TNetNullControl& control) const
Return whether two null controls are the same.

```

```

void AllowedMovements(LaneCollection& lanes, const TNetLink&
                      fromlink, const TNetLane& fromlane, const TNetLink&
                      tolink)
void AllowedMovements(LaneCollection& lanes, const TNetLink&
                      fromlink, const TNetLink& tolink, const TNetLane&
                      tolane)
void AllowedMovements(LaneCollection& lanes, const TNetLink&
                      fromlink, const TNetLink& tolink, bool phase = FALSE)
void InterferingLanes(LaneCollection& lanes, const TNetLane&
                      fromlane, const TNetLane& tolane, bool phase = FALSE)
TNetTrafficControl::ETrafficControl GetVehicleControl(const
                                                       TNetLane& lane) const
Provide definitions for pure virtual functions to throw exceptions.

```

## 2.2.21 TNetIsolatedControl

An isolated signal requires no coordination. It defers to the signalized control for operations.

```

TNetIsolatedControl()
TNetIsolatedControl(NetCoordinatorId id )
TNetIsolatedControl(const TNetIsolatedControl& control)
Construct an isolated control.

TNetIsolatedControl& operator=(const TNetIsolatedControl&
                               control)
Assign an isolated traffic control.

bool operator==(const TNetIsolatedControl& control) const
bool operator!=(const TNetIsolatedControl& control) const
Return whether two isolated controls are the same.

void UpdateSignalizedControl(REAL sim_time)
Isolated control requires no coordination.

```

## 2.2.22 TNetUnsignalizedControl

An unsignalized control specifies the sign control at a node. Some type of sign control is associated with each link attached to an unsignalized intersection. Example values are stop, yield, and no control on this link.

```

TNetUnsignalizedControl()
TNetUnsignalizedControl(TNetNode& node)
TNetUnsignalizedControl(const TNetUnsignalizedControl& control)
Construct an unsignalized traffic control.

TNetUnsignalizedControl(TNetUnsignalizedControlReader& reader,
                       TNetNetwork& network)
Construct an unsignalized traffic control using the reader.

```

```

TNetUnsignalizedControl& operator=(const
    TNetUnsignalizedControl& control)
Assign an unsignalized traffic control.

bool operator==(const TNetUnsignalizedControl& control) const
bool operator!=(const TNetUnsignalizedControl& control) const
Return whether two unsignalized controls are the same.

TNetTrafficControl::ETrafficControl GetVehicleControl(const
    TNetLane& lane) const
Return the vehicle control signal for the specified lane.

void SetVehicleControl(TNetLink& link,
    TNetTrafficControl::ETrafficControl)
Define the vehicle control sign.

void SetVehicleControl(TNetUnsignalizedControlReader& reader,
    TNetNetwork& network)
Define the vehicle control sign using the reader.

```

## 2.2.23 TNetUnsignalizedControlReader

This reader reads unsignalized control values from the database. Each unsignalized control reader has a database table accessor and database fields.

```
TNetUnsignalizedControlReader(TNetReader& reader)
Construct an unsignalized control reader for a given network.
```

```
TNetUnsignalizedControlReader(const
    TNetUnsignalizedControlReader& reader)
Construct a copy of the given unsignalized control reader.
```

```
TNetUnsignalizedControlReader& operator=(const
    TNetUnsignalizedControlReader& reader)
Make the reader a copy of the given unsignalized control reader.
```

```
void Reset()
Reset the iteration over the table.
```

```
void GetNextNode()
Get the next node in the table.
```

```
bool MoreNodes()
Return whether there are any more nodes in the table.
```

```
NetNodeId GetNode() const
Return the id for the current node.
```

```
NetLinkId GetInlink() const
Return the id for incoming link.
```

```
string GetSign() const
    Return the sign control indication on incoming link.
```

## 2.2.24 TNetSignalizedControl

A signalized control specifies the signal phases and phasing plan at a node. A signalized control has a sequence of phases, a phasing plan, a phasing plan offset, a coordinator, and a current phase.

```
TNetSignalizedControl()
TNetSignalizedControl(TNetNode& node)
TNetSignalizedControl(const TNetSignalizedControl& control)
    Construct a signalized traffic control.

TNetSignalizedControl(TNetSignalizedControlReader& reader,
                      TNetNetwork& network)
    Construct a signalized traffic control using the reader.

virtual void AllowedMovements (LaneCollection& lanes, const
                                TNetLink& fromlink, const TNetLane& fromlane, const
                                TNetLink& tolink)
    Return the lanes on next link to which transition from specified lane on this link can be
    made. Return an empty collection if transition is not possible.

virtual void AllowedMovements (LaneCollection& lanes, const
                                TNetLink& fromlink, const TNetLink& tolink, const
                                TNetLane& tolane)
    Return the lanes on this link from which transition to specified lane on next link can be
    made. Return an empty collection if transition is not possible.

virtual void AllowedMovements (LaneCollection& lanes, const
                                TNetLink& fromlink, const TNetLink& tolink, bool
                                phase = FALSE)
    Return the lanes ordered from median that may be used to transition from current link to
    next link.

virtual void InterferingLanes (LaneCollection& lanes, const
                               TNetLane& fromlane, const TNetLane& tolane, bool
                               phase = FALSE)
    Return the lanes that must be examined for interference when transitioning from current
    lane to next lane.

virtual TNetTrafficControl::ETrafficControl
        GetVehicleControl(const TNetLane&) const
    Return the vehicle control signal for the specified lane.

virtual void UpdateSignalizedControl (REAL sim_time)
    Update the traffic control state based on current simulation time.
```

```

virtual TNetPhasingPlan& GetPhasingPlan()
virtual const TNetPhasingPlan& GetPhasingPlan() const
    Return the current phasing plan.

virtual REAL GetPhasingPlanOffset() const
    Return the current phasing plan offset.

virtual void SetPhasingPlan(TNetPhasingPlan& plan)
    Define the phasing plan.

virtual void SetPhasingPlanOffset(REAL offset)
    Define the phasing plan offset.

virtual TNetSignalCoordinator& GetCoordinator()
virtual const TNetSignalCoordinator& GetCoordinator() const
    Return the signalized control coordinator for this signal.

virtual void SetCoordinator(TNetSignalCoordinator& coordinator)
    Define the signalized control coordinator for this signal.

virtual TNetPhase& CreatePhase(TNetPhaseDescription&
    description)
Create a phase.

virtual PhaseCollection& GetPhases()
virtual const PhaseCollection& GetPhases() const
    Return the phases for this signal.

virtual TNetPhase& GetPhase()
virtual const TNetPhase& GetPhase() const
    Return the current phase.

virtual void SetPhase(TNetPhase& phase)
    Update the current phase.

virtual void InitPhase(TNetPhase& phase)
    Initialize the signal to specified phase.

```

## 2.2.25 TNetSignalizedControlReader

This reader reads signalized control values from the database. Each signalized control reader has a database table accessor and database fields.

```

TNetSignalizedControlReader(TNetReader& reader)
    Construct a signalized control reader for a given network.

TNetSignalizedControlReader(const TNetSignalizedControlReader&
    reader)
Construct a copy of the given signalized control reader.

```

```

TNetSignalizedControlReader& operator=(const
    TNetSignalizedControlReader& reader)
    Make the reader a copy of the given signalized control reader.

void Reset()
    Reset the iteration over the table.

void GetNextNode()
    Get the next node in the table.

bool MoreNodes()
    Return whether there are any more nodes in the table.

NetNodeId GetNode() const
    Return the id for the current node.

string GetType() const
    Return the type of the signal.

NetPlanId GetPlan() const
    Return the timing plan id.

REAL GetOffset() const
    Return the offset for coordinated signals.

string GetStarttime() const
    Return the starting time for the plan.

```

## 2.2.26 TNetTimedControl

A timed control specifies the performance of a pre-timed signal. Each timed control has a cycle length and times at which each phase ends.

```

TNetTimedControl()
TNetTimedControl(TNetNode& node)
TNetTimedControl(const TNetTimedControl& control)
    Construct a timed signalized traffic control.

TNetTimedControl(TNetSignalizedControlReader& reader,
                 TNetNetwork& network)
    Construct a timed signalized traffic control using the reader.

TNetTimedControl& operator=(const TNetTimedControl& control)
    Assign a timed control.

bool operator==(const TNetTimedControl& control) const
bool operator!=(const TNetTimedControl& control) const
    Return whether two timed controls are the same.

```

```
virtual void UpdateSignalizedControl(REAL sim_time)
    Update the traffic control state according to algorithm for fixed time controllers.
```

```
virtual TNetPhase& CreatePhase(TNetPhaseDescription&
                                description)
    Create a phase.
```

## 2.2.27 TNetPhase

A phase is a portion of a traffic signal cycle when the allowed movements are unchanged. A phase is composed of intervals where the traffic displays are constant. Each phase has a sequence of intervals, an associated phase description, and one or more next phases to which it can transition.

```
enum EInterval {kGreen, kYellow, kRed}
    Each phase has three intervals.
```

```
TNetPhase()
TNetPhase(TNetPhaseDescription& description)
TNetPhase(const TNetPhase& phase)
    Construct a phase.
```

```
TNetPhase& operator=(const TNetPhase& phase)
    Assign a phase.
```

```
bool operator==(const TNetPhase&) const
bool operator!=(const TNetPhase&) const
    Return whether two phases are the same.
```

```
TNetPhaseDescription& GetPhaseDescription()
const TNetPhaseDescription& GetPhaseDescription() const
    Return phase description associated with this phase.
```

```
TNetPhase::EInterval GetInterval() const
    Return current interval.
```

```
void SetInterval(EInterval interval)
    Update signal interval.
```

```
PhaseCollection& GetNextPhases()
const PhaseCollection& GetNextPhases() const
    Return phases to which this phase can transition.
```

```
void SetNextPhase(TNetPhase& phase)
    Define the next phase.
```

## 2.2.28 TNetPhaseDescription

A phase description specifies the interval lengths and allowed movements and associated turn protections during a phase. Each phase description has a phase number, a minimum green interval length (or green interval length for fixed time), a maximum green interval length (undefined for

fixed time), a green interval extension increment (which equals zero for fixed time), a yellow interval length, a red clearance interval length, and movements allowed during the phase.

```
enum EProtection {kPermitted, kProtected}
    Turn protections are a subset of ETrafficControl.

TNetPhaseDescription(NetPhaseNumber phase)
TNetPhaseDescription(const TNetPhaseDescription& description)
    Construct a phasing plan description.

TNetPhaseDescription& operator=(const TNetPhaseDescription&
    description)
    Assign a phasing plan description.

bool operator==(const TNetPhaseDescription& description) const
bool operator!=(const TNetPhaseDescription& description) const
    Return whether two phase descriptions are the same.

NetPhaseNumber GetPhaseNumber() const
    Return the phase number.

REAL GetGreenLength() const
    Return the green interval length.

REAL GetMinGreenLength() const
    Return the minimum green interval length.

REAL GetMaxGreenLength() const
    Return the maximum green interval length.

REAL GetExtGreenLength() const
    Return the green extension interval length.

REAL GetYellowLength() const
    Return the yellow interval length.

REAL GetRedLength() const
    Return the red clearance interval length.

LinkMovementMap& GetLinkMovements()
const LinkMovementMap& GetLinkMovements() const
    Return all link movements.

LinkProtectionMap& GetLinkMovements(const TNetLink& link)
const LinkProtectionMap& GetLinkMovements(const TNetLink& link)
    const
    Return the link movements for the specified link.
```

```
void SetLengths(REAL min, REAL max, REAL ext, REAL yellow, REAL
    red)
Set the interval lengths.
```

```
void SetLinkMovements(TNetLink& inlink, TNetLink& outlink,
    TNetPhaseDescription::EProtection protection)
Set link movements.
```

## 2.2.29 TNetPhasingPlan

A phasing plan is composed of a series of phase descriptions. Each phasing plan has an id and a sequence of interval length descriptions, one for each phase.

```
TNetPhasingPlan()
TNetPhasingPlan(NetPlanId id)
TNetPhasingPlan(const TNetPhasingPlan& plan)
Construct a phasing plan.
```

```
TNetPhasingPlan& operator=(const TNetPhasingPlan& plan)
Assign a phasing plan.
```

```
bool operator==(const TNetPhasingPlan& plan) const
bool operator!=(const TNetPhasingPlan& plan) const
Return whether two phasing plans are the same.
```

```
PhaseDescriptionCollection& GetPhaseDescriptions()
const PhaseDescriptionCollection& GetPhaseDescriptions() const
Return this phasing plan.
```

```
TNetPhaseDescription& CreatePhaseDescription(NetPhaseNumber
    phase)
Create a phase description.
```

```
NetPlanId GetId() const
Return the plan id.
```

## 2.2.30 TNetPhasingPlanReader

This reader reads phasing plan values from the database. Each phasing plan has a database table accessor and database fields.

```
TNetPhasingPlanReader(TNetReader& reader)
Construct a phasing plan reader for a given network.
```

```
TNetPhasingPlanReader(const TNetPhasingPlanReader& reader)
Construct a copy of the given phasing plan reader.
```

```
TNetPhasingPlanReader& operator=(const TNetPhasingPlanReader&
    reader)
Make the reader a copy of the given phasing plan reader.
```

```

void Reset()
    Reset the iteration over the table.

void GetNextNode()
    Get the next node in the table.

bool MoreNodes()
    Return whether there are any more nodes in the table.

NetNodeId GetNode() const
    Return the id for the current node.

NetPlanId GetPlan() const
    Return the timing plan id.

NetPhaseNumber GetPhase() const
    Return the phase number.

NetLinkId GetInlink() const
    Return the incoming link id.

NetLinkId GetOutlink() const
    Return the outgoing link id.

string GetProtection() const
    Return the turn protection indicator.

```

### **2.2.31 TNetTimingPlanReader**

This reader reads timing plan values from the database. Each timing plan reader has a database table accessor and database fields.

```

TNetTimingPlanReader(TNetReader& reader)
    Construct a timing plan reader for a given network.

TNetTimingPlanReader(const TNetTimingPlanReader& reader)
    Construct a copy of the given timing plan reader.

TNetTimingPlanReader& operator=(const TNetTimingPlanReader&
    reader)
    Make the reader a copy of the given timing plan reader.

void Reset()
    Reset the iteration over the table.

void GetNextPlan()
    Get the next plan in the table.

```

```

bool MorePlans()
    Return whether there are any more plans in the table.

NetPlanId GetPlan() const
    Return the timing plan id.

NetPhaseNumber GetPhase() const
    Return the phase number.

string GetNextPhases() const
    Return the phase numbers of the next phases.

REAL GetGreenMin() const
    Return the minimum length of green interval.

REAL GetGreenMax() const
    Return the maximum length of green interval.

REAL GetGreenExt() const
    Return the length of green interval extension.

REAL GetYellow() const
    Return the length of yellow interval.

REAL GetRedClear() const
    Return the length of red clearance interval.

```

### **2.2.32 TNetSignalCoordinator**

A signal coordinator coordinates the operation of several traffic signals. Each coordinator has a unique id, one or more signalized controllers, and a group of phasing plans that it can tell its controllers to use.

```

TNetSignalCoordinator()
TNetSignalCoordinator(NetCoordinatorId id)
TNetSignalCoordinator(const TNetSignalCoordinator& coordinator)
    Construct a signalized control coordinator.

virtual void UpdateSignalizedControl(REAL sim_time)
    Coordinate signal controls when necessary and then call the
    UpdateSignalizedControl method for each of the controllers.

PhasingPlanSet& GetPhasingPlans()
const PhasingPlanSet& GetPhasingPlans() const
    Return the coordinator's phasing plans.

```

```

TNetPhasingPlan& CreatePhasingPlan(NetNodeId node, NetPlanId
    phasing, TNetTimingPlanReader& timing,
    TNetPhasingPlanReader& reader, TNetNetwork& network,
    PhaseNumberMap& numbers)
Define the phasing plan.

TNetPhasingPlan& GetPhasingPlan(NetPlanId id)
const TNetPhasingPlan& GetPhasingPlan(NetPlanId id) const
Return the phasing plan with specified id.

ControllerCollection& GetControllers()
const ControllerCollection& GetControllers() const
Return the signalized controls coordinated by this coordinator.

void SetController(TNetSignalizedControl& controller)
Define a controller.

NetCoordinatorId GetId() const
Return the coordinator's id.

```

### 2.2.33 TNetSimulationArea

The simulation area describes the simulation region of interest. A simulation area is described by links and whether the link is in the buffer portion of the simulation area.

```

enum EType {kStudy, kBuffer}
Link type indicates whether the link is in the buffer or study area.

TNetSimulationArea(TNetSimulationAreaLinkReader& reader)
TNetSimulationArea(const TNetSimulationArea& area)
Construct a simulation area.

TNetSimulationArea& operator=(const TNetSimulationArea& area)
Assign a simulation area.

bool operator==(const TNetSimulationArea& area) const
bool operator!=(const TNetSimulationArea& area) const
Return whether two simulation areas are the same.

bool IsInSimulationArea(NetLinkId id)
Return whether a link is in the simulation area (study area + buffer).

bool IsInStudyArea(NetLinkId id)
Return whether a link is in the study area.

bool IsInBufferArea(NetLinkId id)
Return whether a link is in the simulation area buffer region.

```

```
LinkIdMap& GetLinks()
const LinkIdMap& GetLinks() const
    Return all links in simulation area.
```

### 2.2.34 TNetSimulationAreaReader

A simulation area reader reads a simulation area from the database. Each reader has a simulation area database table accessor and database fields.

```
TNetSimulationAreaReader(TDbTable areaTable)
Construct a reader for the specified tables.
```

```
TDbTable& GetSimulationAreaTable()
Return the simulation area table.
```

### 2.2.35 TNetSimulationAreaLinkReader

A simulation area link reader reads link values from the database. Each simulation area link reader has a database table accessor and database fields.

```
TNetSimulationAreaLinkReader(TNetSimulationAreaReader& reader)
Construct a simulation area link reader.
```

```
TNetSimulationAreaLinkReader(const TNetSimulationAreaLinkReader&
                             reader)
Construct a copy of the given simulation area link reader.
```

```
TNetSimulationAreaLinkReader& operator=(const
                                         TNetSimulationAreaLinkReader& reader)
Make the reader a copy of the given simulation area link reader.
```

```
void Reset()
Reset the iteration over the table.
```

```
void GetNextLink()
Get the next link in the table.
```

```
bool MoreLinks() const
Return whether there are any more links in the table.
```

```
LinkIdMap GetLinks()
Return the links in the simulation area.
```

### 2.2.36 TGeoPoint

A geographic point contains the coordinates of a position on a map. Each point has an  $x$  coordinate and a  $y$  coordinate.

```

TGeoPoint(REAL x = 0, REAL y = 0)
    Construct a point with the given x and y coordinates.

TGeoPoint(const TGeoPoint& point)
    Construct a copy of the given point.

TGeoPoint& operator=(const TGeoPoint& point)
    Make the point a copy of the given point.

REAL GetX() const
    Return the x coordinate.

REAL GetY() const
    Return the y coordinate.

REAL GetAngleTo(const TGeoPoint& point) const
    Return the angle to the specified point.

```

### 2.2.37 TGeoRectangle

A geographic rectangle is a rectangle in a map coordinate system. Each rectangle has a minimum corner and a maximum corner.

```

TGeoRectangle(const TGeoPoint& corner1, const TGeoPoint&
              corner2)
    Construct a rectangle with the given corners.

TGeoRectangle(const TGeoRectangle& rectangle)
    Construct a copy of the given rectangle.

TGeoRectangle& operator=(const TGeoRectangle& rectangle)
    Make the rectangle a copy of the given rectangle.

void GetCorners(TGeoPoint& corner1, TGeoPoint& corner2) const
    Return the corners.

bool Contains(const TGeoPoint& point) const
    Return whether the rectangle contains the given point.

```

### 2.2.38 TGeoFilterFunction

A geographic filter function selects geographic points. This abstract class must be subclassed to be used.

```

virtual bool operator()(const TGeoPoint& point) const
    Return whether a point is acceptable.

```

## 2.2.39 TGeoFilterNone

This geographic filter accepts all points.

```
TGeoFilterNone(const TGeoFilterNone& filter)
    Construct a copy of the given filter function.

TGeoFilterNone& operator=(const TGeoFilterNone& filter)
    Make the filter a copy of the given filter function.

bool operator()(const TGeoPoint& point) const
    Return whether a point is acceptable.
```

## 2.2.40 TGeoFilterRectangle

This geographic filter accepts all points within a rectangle. Each rectangular filter has a rectangle.

```
TGeoFilterRectangle(const TGeoRectangle& rectangle)
    Construct a rectangular filter for the given rectangle.

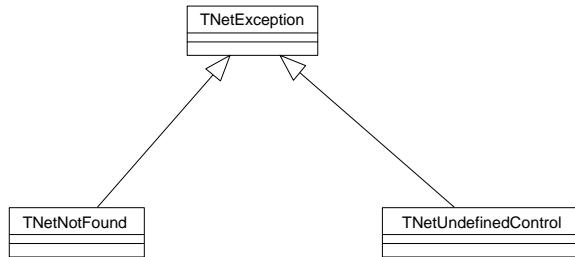
TGeoFilterRectangle(const TGeoFilterRectangle& filter)
    Construct a copy of the given filter function.

TGeoFilterRectangle& operator=(const TGeoFilterRectangle&
    filter)
    Make the filter a copy of the given filter function.

bool operator()(const TGeoPoint& point) const
    Return whether a point is acceptable.
```

## 2.2.41 TNetException

A network exception signals the failure of a network subsystem function. Each exception has a message. Figure 7 shows the hierarchy of exception classes.



**Figure 7: Exception Hierarchy for the TRANSIMS Network Subsystem (unified notation)**

```
TNetException(const string& message = "Network error.")
    Construct an exception with the specified message text.
```

```
TNetException(const TNetException& exception)
    Construct a copy of the given exception.

TNetException& operator=(const TNetException& exception)
    Make the exception a copy of the given exception.

const string& GetMessage() const
    Return the message text for the exception.

class TNetNotFound
    This exception is thrown when an attempt is made to access something that cannot be
    found.

class TNetUndefinedControl
    This exception is thrown when an attempt is made to access the member functions of a null
    control.
```

## **3. IMPLEMENTATION**

### **3.1 C++ Libraries**

The Booch Components [RW 94] provide C++ container classes that the network representation subsystem uses extensively. The subsystem also uses the standard C++ library [Pl 95], the standard C library [Pl 92], and the POSIX library [Ga 95b]. All of these libraries compile on a wide variety of platforms (UNIX and otherwise).

### **3.2 Sources for Traffic Engineering Information**

References [MM 84] and [PP 93] provide an overview of traffic engineering practice incorporated into the TRANSIMS network representation.

## 4. USAGE

### 4.1 Accessing Network Data via C++

The following example shows how to create a network from data tables and to retrieve the node and link objects in the network:

```
// Open the data directory.  
TDbDirectory directory(TDbDirectoryDescription("IOC-1"));  
  
// Open the network data sources.  
TDbSource nodeSource(directory, directory.GetSource("Node"));  
TDbSource linkSource(directory, directory.GetSource("Link"));  
TDbSource pocketSource(directory,  
    directory.GetSource("Pocket Lane"));  
TDbSource parkingSource(directory,  
    directory.GetSource("Parking"));  
TDbSource laneSource(directory,  
    directory.GetSource("Lane Connectivity"));  
TDbSource ucontrolSource(directory,  
    directory.GetSource("Unsignalized Node"));  
TDbSource scontrolSource(directory,  
    directory.GetSource("Signalized Node"));  
TDbSource phasingSource(directory,  
    directory.GetSource("Phasing Plan"));  
TDbSource timingSource(directory,  
    directory.GetSource("Timing Plan"));  
  
// Read the table names from standard input and open them.  
char line[80];  
cin.getline(line, 80);  
TDbTable nodeTable(nodeSource, nodeSource.GetTable(line));  
cin.getline(line, 80);  
TDbTable linkTable(linkSource, linkSource.GetTable(line));  
cin.getline(line, 80);  
TDbTable pocketTable(pocketSource, pocketSource.GetTable(line));  
cin.getline(line, 80);  
TDbTable parkingTable(parkingSource,  
    parkingSource.GetTable(line));  
cin.getline(line, 80);  
TDbTable laneTable(laneSource, laneSource.GetTable(line));  
cin.getline(line, 80);  
TDbTable ucontrolTable(ucontrolSource,  
    ucontrolSource.GetTable(line));  
cin.getline(line, 80);  
TDbTable scontrolTable(scontrolSource,  
    scontrolSource.GetTable(line));  
cin.getline(line, 80);  
TDbTable phasingTable(phasingSource,  
    phasingSource.GetTable(line));  
cin.getline(line, 80);  
TDbTable timingTable(timingSource, timingSource.GetTable(line));  
  
// Create a network reader for the tables.  
TNetReader reader(nodeTable, linkTable, pocketTable, parkingTable,
```

```

        laneTable, ucontrolTable, scontrolTable, phasingTable,
        timingTable);

    // Create the network.
    TNetNetwork network;

    // Read all of the network data.
    TNetSubnetwork subnetwork(reader, network);

    // Get the nodes and links.
    TNetSubnetwork::NodeSet& nodes = subnetwork.GetNodes();
    TNetSubnetwork::LinkSet& links = subnetwork.GetLinks();

```

## 4.2 Network Data Tables

Ten data tables are required to describe a TRANSIMS format road network. The tables, their fields, and an example are shown below.

### 4.2.1 File Formats

The preferred format for data files is ASCII, with columns delimited by tab characters. Records are terminated by an end-of-line character. Formats such as dBASE III+ are also acceptable. Table 1 through Table 10 below define the fields for the network data tables.

**Table 1: Node Table Format**

Column Name	Description	Allowed Values
ID	ID # of the node	Integer: 1 through 4,294,967,295
ABSCISSA	X-coordinate of the node	Floating-point number
ORDINATE	Y-coordinate of the node	Floating-point number

**Table 2: Link Table Format**

Column Name	Description	Allowed Values
ID	ID # of the link	Integer: 1 through 4,294,967,295
NODEA	ID # of the node at A	Integer: 1 through 4,294,967,295
NODEB	ID # of the node at B	Integer: 1 through 4,294,967,295
PERMLANESA	Number of lanes on the link heading toward node A, not including pocket lanes	Integer: 1 through 255
PERMLANESB	Number of lanes on the link heading toward node B, not including pocket lanes	Integer: 1 through 255
LEFTPCKTSA	Number of pocket lanes to the left of the permanent lanes heading toward node A	Integer: 1 through 255
LEFTPCKTSB	Number of pocket lanes to the left of the permanent lanes heading toward node B	Integer: 1 through 255
RGHTPCKTSA	Number of pocket lanes to the right of the permanent lanes heading toward node A	Integer: 1 through 255
RGHTPCKTSB	Number of pocket lanes to the right of the permanent lanes heading toward node B	Integer: 1 through 255
TWOWAYTURN	Whether there is a two-way left-turn lane in the center of the link	One character: ‘F’ = false/no ‘T’ = true/yes
LENGTH	Length of the link	Floating-point number
GRADE	Percent grade from node A to node B, uphill being a positive number	Floating-point number
SETBACKA	Set-back distance from the center of the intersection at node A	Floating-point number
SETBACKB	Set-back distance from the center of the intersection at node B	Floating-point number
CAPACITYA	Total capacity (in vehicles-per-hour) for the lanes traveling to node A	Floating-point number
CAPACITYB	Total capacity (in vehicles-per-hour) for the lanes traveling to node B	Floating-point number
SPEEDLMTA	Speed limit for vehicles traveling toward node A	Floating-point number
SPEEDLMTB	Speed limit for vehicles traveling toward node B	Floating-point number
FREESPDA	Free-flow speed for vehicles traveling toward node A	Floating-point number

Column Name	Description	Allowed Values
ID	ID # of the link	Integer: 1 through 4,294,967,295
NODEA	ID # of the node at A	Integer: 1 through 4,294,967,295
FREESPDB	Free-flow speed for vehicles traveling toward node B	Floating-point number
CRAWLSPDA	Crawl speed for vehicles traveling toward node A	Floating-point number
CRAWLSPDB	Crawl speed for vehicles traveling toward node B	Floating-point number
FUNCTCLASS	Functional class of the link	Ten characters
COSTA	Travel cost for vehicles traveling toward node A	Integer: 0 through 4,294,967,295
COSTB	Travel cost for vehicles traveling toward node B	Integer: 0 through 4,294,967,295
THRUA	Default through link connected at node A	Integer: 1 through 4,294,967,295
THRUB	Default through link connected at node B	Integer: 1 through 4,294,967,295

**Table 3: Pocket Lane Table Format\***

Column Name	Description	Allowed Values
ID	ID # of the pocket lane accessory	Integer: 1 through 4,294,967,295
NODE	ID # of the node into which the pocket lane leads	Integer: 1 through 4,294,967,295
LINK	ID # of the link on which the pocket lane lies	Integer: 1 through 4,294,967,295
OFFSET	Starting position of the pocket lane, measured from the node away from which it is traveling (pull-out pockets only)	Floating-point number
LANE	Lane number of the pocket lane	Integer: 1 through 255
STYLE	Type of the pocket lane	One character: ‘T’ = turn pocket ‘P’ = pull-out pocket ‘M’ = merge pocket
LENGTH	Length of the pocket lane (turn pockets and merge pockets always start or end at the appropriate limit line)	Floating-point number

---

\* Note that pocket lanes are accessories and so have an accessory id; this id must be unique over all types of accessories, not just pocket lane accessories.

**Table 4: Parking Table Format\***

Column Name	Description	Allowed Values
ID	ID # of the parking place	Integer: 1 through 4,294,967,295
NODE	ID # of the node into which the parking leads	Integer: 1 through 4,294,967,295
LINK	ID # of the link on which the parking lies	Integer: 1 through 4,294,967,295
OFFSET	Starting position of the parking, measured from the node away from which it is traveling	Floating-point number
STYLE	Type of the parking	Five characters: ‘PRSTR’ = parallel on street ‘HISTR’ = head in on street ‘DRVWY’ = driveway ‘LOT’ = parking lot ‘BNDRY’ = network boundary
CAPACITY	Number of vehicles the parking can accommodate, zero for unlimited capacity	Integer: 0 through 65,535
GENERIC	Whether the accessory represents generic parking (not an actual driveway, lot, etc., but a group/aggregate of them used to simplify modeling)	One character ‘T’ = true/yes ‘F’ = false/no

**Table 5: Lane Connectivity Table Format**

Column Name	Description	Allowed Values
NODE	ID # of the node	Integer: 1 through 4,294,967,295
INLINK	ID # of an incoming link	Integer: 1 through 4,294,967,295
INLANE	Lane number of an incoming lane	Integer: 1 through 255
OUTLINK	ID # of an outgoing link	Integer: 1 through 4,294,967,295
OUTLANE	Lane number of an outgoing lane	Integer: 1 through 255

\* Note that parking areas are accessories and so have an accessory id; this id must be unique over all of types of accessories, not just parking area accessories.

**Table 6: Unsignalized Node Table Format**

Column Name	Description	Allowed Values
NODE	ID # of the node	Integer: 1 through 4,294,967,295
LINK	ID # of an incoming link	Integer: 1 through 4,294,967,295
SIGN	Type of sign control on link	One character: ‘S’ = stop ‘Y’ = yield ‘N’ = none

**Table 7: Signalized Node Table Format**

Column Name	Description	Allowed Values
NODE	ID # of the node	Integer: 1 through 4,294,967,295
TYPE	Type of the signal	One character ‘T’ = timed ‘A’ = actuated
PLAN	ID # of a timing plan	Integer: 1 through 255
OFFSET	Offset in seconds for coordinated signals	Floating-point number
STARTTIME	Starting time for this plan	A character string with the day of week (‘SUN’ = Sunday, ‘MON’ = Monday, ‘TUE’ = Tuesday, ‘WED’ = Wednesday, ‘THU’ = Thursday, ‘FRI’ = Friday, ‘SAT’ = Saturday, ‘WKE’ = any weekend day, ‘WKD’ = any weekday, ‘ALL’ = any day) followed by the time of day (on a 24-hour clock). For example, ‘WKD13:20’ is any weekday at 1:20 in the afternoon

**Table 8: Phasing Plan Table Format**

Column Name	Description	Allowed Values
NODE	ID # of the node	Integer: 1 through 4,294,967,295
PLAN	ID # of a timing plan	Integer: 1 through 255
PHASE	Phase number	Integer: 1 through 255
LINK	ID # of an incoming link	Integer: 1 through 4,294,967,295
OUTLINK	ID # of an outgoing link	Integer: 1 through 4,294,967,295
PROTECTION	Turn protection indicator	One character: ‘P’ = protected ‘U’ = unprotected

**Table 9: Timing Plan Table Format<sup>\*</sup>**

Column Name	Description	Allowed Values
PLAN	ID # of a timing plan	Integer: 1 through 255
PHASE	Phase number	Integer: 1 through 255
NEXTPHASES	Phase number(s) of the next phase(s) in sequence	String of phase numbers, separated by slashes
GREENMIN	Minimum length in seconds of green interval, or fixed green length for timed signal	Floating-point number
GREENMAX	Maximum length in seconds of green interval	Floating-point number
GREENEXT	Length in seconds of green extension interval	Floating-point number
YELLOW	Length in seconds of yellow interval	Floating-point number
REDCLEAR	Length in seconds of red clearance interval	Floating-point number

**Table 10: Study Area Link Table Format**

Column Name	Description	Allowed Values
ID	ID # of the link	Integer: 1 through 4,294,967,295
BUFFER	Whether the link is in the buffer area or the study area	One character: ‘Y’ = in buffer area ‘N’ = in study area

#### 4.2.2 Example

An example network is depicted in Figure 8, followed by Table 11 through Table 22 which illustrate the data tables that correspond to this network. This example shows how the following are represented in this specification:

- signalized intersection
- unsignalized intersection
- a node without an intersection
- turn pocket
- merge pocket
- pull-out pocket
- lane use
- lane connectivity
- phasing plan
- timing plan

---

\* GREENMAX and GREENEXT are undefined for pre-timed signals and should be indicated in the table as 0.0. REDCLEAR should be specified as 0.0 when there is no red clearance interval.

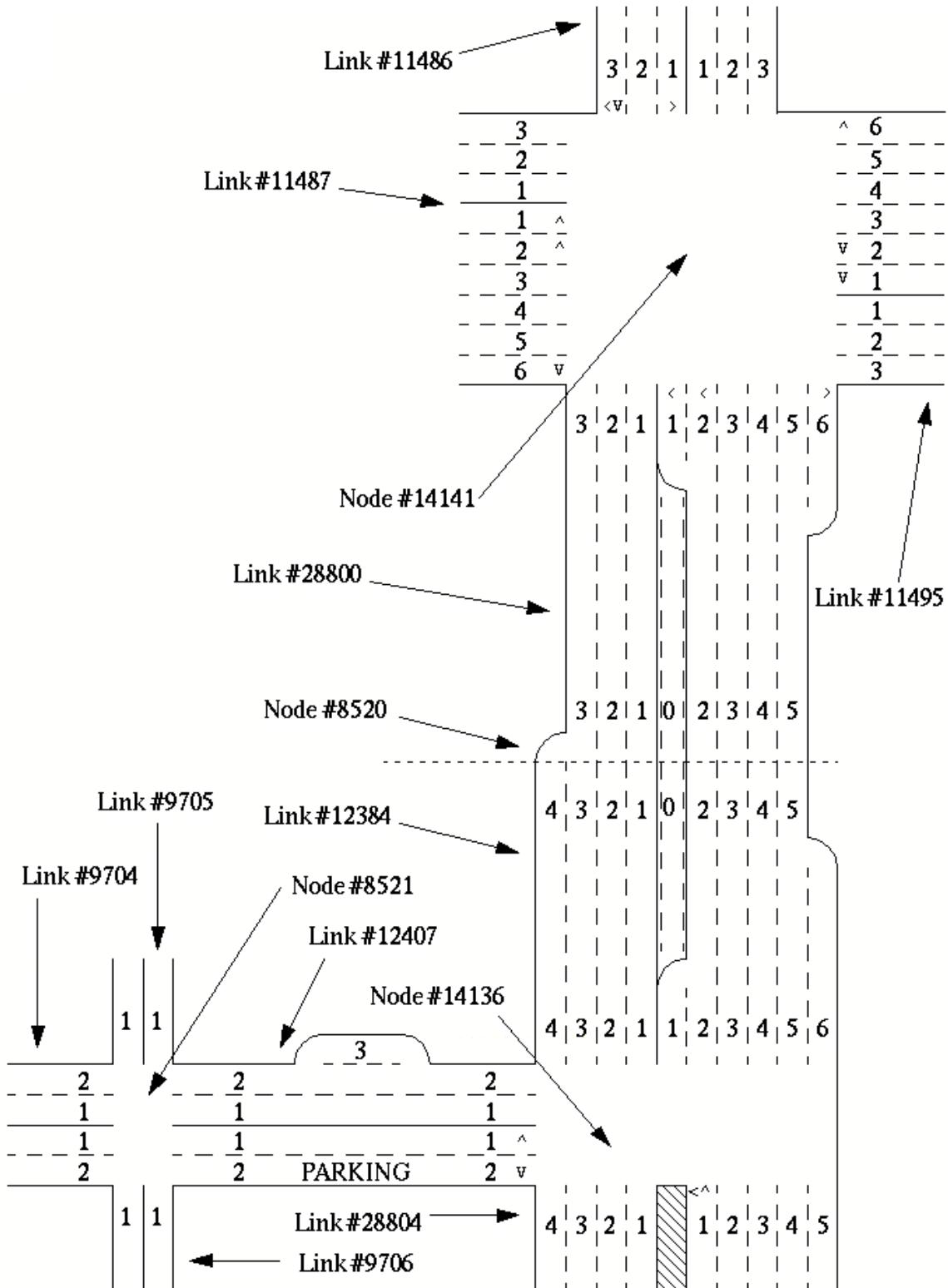


Figure 8: Example Network

Three of the nodes (#8521, #14136, and #14141) in Table 11 are associated with intersections, while the other (#8520) is associated with a change in the permanent number of lanes.

**Table 11: Example Node Table**

ID	ABSCISSA	ORDINATE
8520	1.0	0.5
8521	0.0	0.0
14136	1.0	0.0
14141	1.0	1.0

The three links in lanes are counted and numbered.

Table 12 illustrates how the permanent number of lanes, left pocket lanes, and right pocket lanes are counted and numbered.

**Table 12: Example Link Table (part I)**

ID	NODEA	NODEB	PERMLANESA	PERMLANESB	LEFTPCKTSA	LEFTPCKTSB
12384	14136	8520	4	4	0	1
12407	8521	14136	2	2	0	0
28800	8520	14141	3	4	0	1

**Table 13: Example Link Table (part II)**

RGHTPCKTSA	RGHTPCKTSB	TWOWAYTURN	LENGTH	GRADE	SETBACKA	SETBACKB
0	1	T	1000.0	1.0	15.0	0.0
1	0	F	1200.0	0.0	10.0	15.0
0	1	T	1500.00	1.0	0.0	15.0

**Table 14: Example Link Table (part III)**

CAPACITYA	CAPACITYB	SPEEDLMTA	SPEEDLMTB	FREESPDA	FREESPDB
800	1000	45.0	45.0	50.0	50.0
500	500	35.0	35.0	40.0	40.0
800	1000	45.0	45.0	50.0	50.0

**Table 15: Example Link Table (part IV)**

CRAWLSPDA	CRAWLSPDB	THRU A	THRU B	COSTA	COSTB	FUNCTCLASS
15.0	15.0	28804	28800	1	1	OTHER
10.0	10.0	9704	0	1	1	OTHER
15.0	15.0	12384	11486	1	1	OTHER

All three types of pocket lanes (turn pocket, merge pocket, and pull-out pocket) are represented in Table 16.

**Table 16: Example Pocket Lane Table**

ID	NODE	LINK	OFFSET	LANE	STYLE	LENGTH
85201	8520	12384	0	1	M	100.0
852062	8520	12384	0	6	M	200.0
85213	8521	12407	600.0	3	P	30.0
141411	14141	28800	0	1	T	200.0
141416	14141	28800	0	6	T	300.0

Referring to Figure 8, the first six rows in Table 17 may be understood as follows: Lanes 1 and 2 on link 11487 (attached to node 14141) are exclusive left-turn lanes connecting only to lanes 1 and 2 on link 11486. Lanes 3, 4, and 5 on link 11487 are through lanes to lanes 1, 2, and 3 on link 11495. Lane 6 on link 11487 is a right-turn-only lane connecting to lane 3 on link 28800. Lane 3 on link 11486 connects to both lane 2 on link 28800 and lane 3 on link 11487, as shown in rows 9 and 10 of the table.

**Table 17: Example Lane Connectivity Table**

NODE	INLINK	INLANE	OUTLINK	OUTLANE
14141	11487	1	11486	1
14141	11487	2	11486	2
14141	11487	3	11495	1
14141	11487	4	11495	2
14141	11487	5	11495	3
14141	11487	6	28800	3
14141	11486	1	11495	1
14141	11486	2	28800	1
14141	11486	3	28800	2
14141	11486	3	11487	3
14141	11495	1	28800	1
14141	11495	2	28800	2
14141	11495	3	11487	1
14141	11495	4	11487	2
14141	11495	5	11487	3
14141	11495	6	11486	3
14141	28800	1	11487	1
14141	28800	2	11487	2
14141	28800	3	11486	1
14141	28800	4	11486	2
14141	28800	5	11486	3
14141	28800	6	11495	3
8520	12384	2	28800	2
8520	12384	3	28800	3
8520	12384	4	28800	4
8520	12384	5	28800	5
8520	28800	1	12384	1
8520	28800	2	12384	2
8520	28800	3	12384	3
8520	28800	3	12384	4
14136	12407	1	12384	1
14136	12407	2	28804	4
14136	12384	1	28804	1
14136	12384	2	28804	2
14136	12384	3	28804	3
14136	12384	4	28804	4
14136	12384	4	12407	2
14136	28804	1	12407	1
14136	28804	1	12384	2
14136	28804	2	12384	3
14136	28804	3	12384	4
14136	28804	4	12384	5

NODE	INLINK	INLANE	OUTLINK	OUTLANE
14136	28804	5	12384	6
8521	12407	1	9704	1
8521	12407	1	9706	1
8521	12407	2	9704	2
8521	12407	2	9705	1
8521	9704	1	12407	1
8521	9704	1	9705	1
8521	9704	2	12407	2
8521	9704	2	9706	1
8521	9705	1	9706	1
8521	9705	1	9704	2
8521	9705	1	12407	1
8521	9706	1	9705	1
8521	9706	1	12407	2
8521	9706	1	9704	1

Several types of parking (lot, street, driveway, and generic vs. actual) are represented in Table 18.

**Table 18: Example Parking Table**

ID	NODE	LINK	OFFSET	STYLE	CAPACITY	GENERIC
1001	28800	8520	400	LOT	50	T
1002	12384	14136	300	PRSTR	10	T
1003	12407	14136	200	HISTR	10	T
1004	12407	8521	100	DRVWY	1	F

The number of permanent lanes changes from three lanes on link 28800 to four lanes on link 12384 at node 8520. No right-of-way sign control is required at this node. A stop sign is indicated on link 12407 at node 14136, with no sign control on the other two links at this node. Table 19 illustrates these.

**Table 19: Example Unsignalized Node Table.**

NODE	INLINK	SIGN
8520	12384	N
8520	28800	N
14136	12407	S
14136	12384	N
14136	28804	N

Node 14141 has a pre-timed signal control with an offset of 19.0 seconds. A single timing and phasing plan is always in effect. Node 8521 is defined as having an actuated signal and two timing and phasing plans. Table 20 illustrates these.

**Table 20: Example Signalized Node Table**

NODE	TYPE	PLAN	OFFSET	STARTTIME
14141	T	1	19.0	'ALL0:00'
8521	A	2	0.0	'ALL18:00'
8521	A	3	0.0	'WKD7:00'

The movements permitted during phase 1 at node 14141 are through movements between links 11487 and 11495, as well as right-turn movements from these links. The right turns are protected, and the protection for the through movements is blank because we don't know whether to consider them as protected or unprotected. (This issue was raised earlier.) Additionally, unprotected right turns are permitted from links 11486 and 28800 during phase 1. The first six rows of Table 21 specify this information.

**Table 21: Example Phasing Plan Table**

NODE	PLAN	PHASE	INLINK	OUTLINK	PROTECTION
14141	1	1	11487	11495	U
14141	1	1	11487	28800	P
14141	1	1	11495	11487	U
14141	1	1	11495	11486	P
14141	1	1	11486	11487	U
14141	1	1	28800	11495	U
14141	1	2	11487	28800	P
14141	1	2	11495	11486	P
14141	1	2	11486	11495	P
14141	1	2	28800	11487	P
14141	1	2	28800	11495	U
14141	1	2	11486	11487	U
14141	1	3	11487	28800	P
14141	1	3	28800	11487	P
14141	1	3	28800	11486	U
14141	1	3	11486	11487	U
14141	1	4	11487	28800	P
14141	1	4	11486	11495	U
14141	1	4	11486	28800	U
14141	1	4	11486	11487	P
14141	1	4	28800	11486	U
14141	1	4	28800	11495	P
14141	1	4	11495	11486	U
14141	1	5	11487	11486	P
14141	1	5	11487	28800	P
14141	1	5	11495	28800	P

NODE	PLAN	PHASE	INLINK	OUTLINK	PROTECTION
14141	1	5	11495	11486	U
14141	1	5	11486	11487	P
14141	1	5	28800	11495	P
14141	1	6	11487	28800	P
14141	1	6	11495	28800	P
14141	1	6	11495	11487	U
14141	1	6	11495	11486	P
14141	1	6	11486	11487	U
14141	1	6	28800	11495	P
8521	2	1	9705	9704	U
8521	2	1	9705	9706	U
8521	2	1	9705	12407	U
8521	2	1	9706	9705	U
8521	2	1	9706	12407	U
8521	2	1	9706	9704	U
8521	2	2	12407	9704	U
8521	2	2	12407	9705	U
8521	2	2	12407	9706	U
8521	2	2	9704	12407	U
8521	2	2	9704	9705	U
8521	2	2	9704	9706	U
8521	3	1	9705	9704	U
8521	3	1	9705	9706	U
8521	3	1	9705	12407	U
8521	3	1	9706	9705	U
8521	3	1	9706	12407	U
8521	3	1	9706	9704	U
8521	3	2	12407	9706	P
8521	3	2	9704	9705	P
8521	3	3	12407	9704	U
8521	3	3	12407	9705	U
8521	3	3	12407	9706	U
8521	3	3	9704	12407	U
8521	3	3	9704	9705	U
8521	3	3	9704	9706	U

Plan 1 in Table 22 was specified in Table 20 as applicable to node 14141. This is a timed signal with green, yellow, and red clearance intervals as indicated in row 1 of the table. Plans 2 and 3 for node 8521 were invented as illustrations and may not make sense as real timing plans.

**Table 22: Example Timing Plan Table**

PLAN	PHASE	NEXTPHASES	GREENMIN	GREENMAX	GREENEXT	YELLOW	REDCLEAR
1	1	2	35.0	0.0	0.0	4.0	0.0
1	2	3	5.0	0.0	0.0	3.0	0.0
1	3	4	8.0	0.0	0.0	3.0	0.0
1	4	5	32.0	0.0	0.0	4.0	0.0
1	5	6	9.0	0.0	0.0	3.0	0.0
1	6	1	1.0	0.0	0.0	3.0	0.0
2	1	2	12.0	30.0	4.0	3.0	0.0
2	2	1	10.0	40.0	4.0	3.0	0.0
3	1	2	12.0	30.0	4.0	3.0	1.0
3	2	3	4.0	8.0	2.0	3.0	0.0
3	3	1	10.0	20.0	4.0	3.0	1.0

## **5. FUTURE WORK**

Future work planned for the TRANSIMS network representation subsystem will focus on the inclusion of additional modes of travel (bus, rail, etc.), time-of-day variation in network properties, and more complicated traffic controls (actuated signals and signals coordinated over a wide area). We will also provide a highly portable alternative implementation that is not based on commercial products such as the Booch Components. Performance enhancements are also anticipated.

## 6. REFERENCES

- [FHA 88] Federal Highway Administration, *Manual on Uniform Traffic Control Devices*, (Washington, D.C., 1988).
- [Ga 95b] B. O. Gallmeister, *POSIX.4: Programming for the Real World*, (Sebastopol, California: O'Reilly & Associates, 1995).
- [ITE ] Institute of Transportation Engineers, *Traffic Detector Handbook*, (Washington, D.C., no date given).
- [ITE 93] Institute of Transportation Engineers, *Traffic Control Systems Handbook*, (Washington, D.C. : ITE Pub. No. LP-123, 1985).
- [MM 84] M. D. Meyer and E. J. Miller, *Urban Transportation Planning*, (New York: McGraw-Hill, 1984).
- [Or 93] F. L. Orcutt, Jr., *The Traffic Signal Book*, (Englewood Cliffs, New Jersey: Prentice Hall, 1993).
- [Pl 92] P. J. Plauger, *The Standard C Library*, (Englewood Cliffs, New Jersey: Prentice Hall, 1992).
- [Pl 95] P. J. Plauger, *The Draft Standard C++ Library*, (Englewood Cliffs, New Jersey: Prentice Hall, 1995).
- [PP 93] C. S. Papacostas and P. D. Prevedouros, *Transportation Engineering and Planning*, (Englewood Cliffs, New Jersey: Prentice Hall, 1993).
- [RW 94] Rogue Wave Software, *The C++ Booch Components*, Version 2.3, (Corvallis, Oregon: Rogue Wave Software, 1994).